PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 9/46 — A1

(11) International Publication Number: **WO 98/19239**

(43) International Publication Date: 7 May 1998 (07.05.98)

(21) International Application Number: PCT/US97/19304

(22) International Filing Date: 24 October 1997 (24.10.97)

(30) Priority Data:
740,285 25 October 1996 (25.10.96) US

(71) Applicant: HE HOLDINGS, INC., doing business as HUGHES ELECTRONICS [US/US]; 7200 Hughes Terrace, P.O. Box 80028, Los Angeles, CA 90045-0066 (US).

(72) Inventors: THOMPSON, Christopher, J.; 2770 165th Street, Surrey, British Columbia V4P 2L8 (CA). KRUCHTEN, Phillippe; 2906 West 37th Avenue, Vancouver, British Columbia V6N 2T9 (CA).

(74) Agents: GRUNEBACH, Georgann, S. et al.; Hughes Electronics, 7200 Hughes Terrace, P.O. Box 80028, Los Angeles, CA 90045-0066 (US).

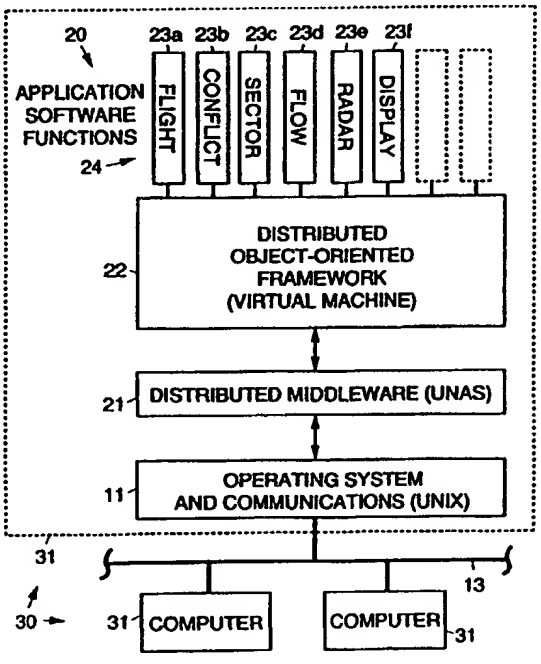
(81) Designated States: CA, JP, KR, NO, European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.

(54) Title: DISTRIBUTED VIRTUAL SOFTWARE INTERFACE OR MACHINE

(57) Abstract

A distributed virtual software interface (20) that provides a reusable framework for building large, distributed fault tolerant Ada applications. The software architecture (20) is implemented in a distributed computer system (30) having a plurality of computers (31) that are interconnected by way of a network (13), and wherein each computer (31) has an operating system. The architecture (20) includes a distributed intermediate software layer (21) that is distributed among the plurality of computers (31) and interfaces with the operating system (11) of the computer (31) on which it is disposed. The distributed intermediate software layer (21) generates intermediate instructions that cause the operating system (11) to implement primitive operating system instructions. The distributed virtual software interface (20) has a distributed object-oriented software layer (22) distributed among the plurality of computers (31) that provides communication between computers (31) using objects. The distributed object-oriented software layer (22) includes instructions that distribute objects of the same class to computers (31) that are linked by attributes, operations and associations between objects within a class in response to the creation of a new object on one of the computers (31). Communication between computers (31) is provided using a number of predefined communication protocols. The distributed object-oriented software layer (22) interfaces with the distributed middle software layer (21) disposed on each respective computer (31) and generates the intermediate instructions. At least one software application is provided on each computer (31) that interfaces with the distributed object-oriented software layer (22) and processes objects routed to it.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece			TR	Turkey
BG	Bulgaria	HU	Hungary	ML	Mali	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MN	Mongolia	UA	Ukraine
BR	Brazil	IL	Israel	MR	Mauritania	UG	Uganda
BY	Belarus	IS	Iceland	MW	Malawi	US	United States of America
CA	Canada	IT	Italy	MX	Mexico	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NE	Niger	VN	Viet Nam
CG	Congo	KE	Kenya	NL	Netherlands	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NO	Norway	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	NZ	New Zealand		
CM	Cameroon			PL	Poland		
CN	China	KR	Republic of Korea	PT	Portugal		
CU	Cuba	KZ	Kazakstan	RO	Romania		
CZ	Czech Republic	LC	Saint Lucia	RU	Russian Federation		
DE	Germany	LI	Liechtenstein	SD	Sudan		
DK	Denmark	LK	Sri Lanka	SE	Sweden		
EE	Estonia	LR	Liberia	SG	Singapore		

DISTRIBUTED VIRTUAL SOFTWARE INTERFACE OR MACHINE

BACKGROUND

The present invention relates generally to distributed, fault tolerant computer systems, and more particularly, to a distributed virtual software interface or machine that interfaces between computers, operating systems and applications that run on the computers of a distributed, multi-computer fault tolerant computer system.

Computers that are used in distributed, fault tolerant computer systems, for example, have an operating system that embodies primitive instructional codes that implement various services. Such primitive instructional codes route data to and from storage devices, write data to display devices, send data to a printer or modem, and transmit data across a network to other computers, for example. Unfortunately, conventional computers and computer operating systems inherently limit the development of large, distributed, fault tolerant computer systems, such as air traffic control systems, and the like.

Conventional computer architectures and operating systems do not directly support distributed computing environments where there is direct cooperation between several processes required to perform specified system functions, including those with client/server relationships. In addition, conventional computer architectures and operating systems are not easily reconfigured on demand or in response to detected hardware or operating system faults so that continuous service is provided in the event of such faults.

There are two primary motivations for building a distributed virtual machine as contemplated by the present invention. The first is to permit large scale software reuse

within a product line (including all of its related parts: process. design, source code, and documentation. The second is separating the functional concerns of the application domain from the requirements of building a reliable distributed computer system.

Accordingly, it is an objective of the present invention to provide for a distributed virtual software interface that interfaces between computers. operating systems and applications that run on the computers of a distributed. multi-computer fault tolerant computer system.

SUMMARY OF THE INVENTION

To meet the above and other objectives, the present invention provides for a distributed virtual software interface that provides a reusable framework for building large, distributed fault tolerant Ada applications. Ada is a registered trademark of the U.S. government. Ada Joint Program Office. The framework. referred to as a distributed virtual machine or distributed virtual software interface. provides a portable suite of integrated services for building a wide variety of distributed. fault tolerant computer systems. The distributed virtual software interface specifically provides a foundation for a family of automated air traffic control systems currently under development by the assignee of the present invention.

The distributed virtual software interface provides for an object-oriented software architecture that is implemented in a distributed computer system having a plurality of computers that are interconnected by way of a network. Each computer has an operating system that implements primitive operating system instructions such as network communications. data storage. data display. and the like.

The architecture comprises a distributed intermediate software layer that is distributed among the plurality of computers and interfaces with the operating system of the computer on which it is disposed and that generates intermediate instructions that cause the operating system to implement primitive operating system instructions in response thereto. A distributed object-oriented software layer is distributed among the plurality of computers providing communication between computers using objects that are instances of object classes that are defined by attributes of objects, operations on objects and associations between objects. The distributed object-oriented software layer comprises instructions that distribute objects of the same class to computers that are linked by the attributes, operations and associations between the objects within the class in response to the creation of a new object on one of the computers. Communication between computers is provided using predefined communication protocols. At least one software application is disposed on each computer that interfaces with the

distributed object-oriented software layer disposed on the respective computer and processes objects distributed to it.

The distributed virtual software interface provides an "abstract instruction set" for developing distributed computer systems, facilitating application development without concern for the actual physical distribution of the computers forming the system. This level of abstraction is essential for managing complex air traffic control systems, for example, which typically require the development of up to one million lines of Ada source code to effectively operate the distributed system.

The architecture of the distributed virtual software interface supports a distributed computing system where there is direct cooperation between processes required to perform specified system functions, including client/server relationships. In addition, distributed computer systems employing the distributed virtual software interface may also be reconfigured on demand or in response to detected faults so that the system provides continuous uninterrupted service.

The architecture of the distributed virtual software interface provides a robust architectural framework for building large distributed Ada-based computer systems. The architecture of the distributed virtual software interface provides a framework for building similar fault tolerant applications through large scale reuse of previously developed blocks of code. Large blocks of the distributed virtual software interface may be reused within an array of computer systems, thus minimizing development costs of additional systems.

BRIEF DESCRIPTION OF THE DRAWINGS

The various features and advantages of the present invention may be more readily understood with reference to the following detailed description taken in conjunction with the accompanying drawings, wherein like reference numerals designate like structural elements, and in which:

Fig. 1 illustrates a conventional software architecture known in the prior art;

Fig. 2 is a diagram showing an object-oriented software architecture implemented in a distributed computer system in accordance with the principles of the present invention;

Fig. 3 shows a static view of a typical software architecture built using the object-oriented software architecture of Fig. 2;

Fig. 4 shows a logical view of a set of classes making use of distributed object services of the object-oriented software architecture;

Fig. 5 shows an illustration of an example object scenario based on the class diagram of Fig. 4, using additional operation annotations;

Fig. 6 shows the same sequence of operations depicted in the scenario view of Fig. 5, but showing an actual distribution of the resulting software;

Fig. 7 illustrates independent processing elements that comprise a class implemented using distributed object services provided by the object-oriented software architecture of Fig. 2; and

Fig. 8 illustrates elements of a distributed class.

DETAILED DESCRIPTION

Referring to the drawing figures, Fig. 1 illustrates a conventional software architecture 10 that may be used with a plurality of computers 14 connected to a network 13. Each of the computers 14 has an operating system 11 that permits communication between the computers 14 and that implements primitive instructions that operates each computer 14. Typically, one or more software applications 12 run on each computer 14. In an example of an air traffic control system, such software applications 12 may include flight management software 12a, conflict prediction software 12b, sector management software 12c, flow management software 12d, radar data processing software 12, and display data processing software 12f, for example.

In the conventional software architecture 10 running on each of the networked computers 14, each software application 12 directly communicates with every other application using low level operating system services. This results in tight coupling between the applications, wherein each application knows of the existence of the other applications. In addition, a large amount of replicated functionality is implemented within each application. Furthermore, each application is dependent on the particular computer operating system and related hardware.

To improve upon the conventional software architecture 10 of Fig. 1, Fig. 2 shows an improved distributed object-oriented architecture 20 that forms a distributed virtual software interface or machine in accordance with the principles of the present invention. The distributed object-oriented framework realized by the distributed virtual software interface allows each of the applications to be less dependent on each other due to its implicit invocation capabilities, allows common functions to be implemented once within the framework rather than implemented repeatedly within the applications, and allows applications to be developed without direct dependency on the operating system or hardware. The distributed object-oriented architecture 20 is implemented in a distributed computer system 30 having a plurality of computers 31 that each have an operating system 11 and that are interconnected by way of a network 13.

The architecture 20 includes a distributed intermediate software layer 21 that is distributed among the plurality of computers 31. At least a portion of the distributed

intermediate software layer 21 is disposed on each computer 31, and interfaces with the operating system 11 of the computer 31 on which it resides. The distributed intermediate software layer 21 causes the operating system 11 to implement primitive operating system instructions in response to intermediate instructions that are generated thereby. A distributed object-oriented software layer 22 is distributed among the plurality of computers 31. At least a portion of the distributed object-oriented software layer 22 is disposed on each computer 31, and interfaces with the respective portion of the distributed middle software layer 21 residing on the computer 31. At least one software application is disposed on each computer 31 that interfaces with the distributed object-oriented software layer 22 disposed on the respective computer 31.

The distributed object-oriented software layer 22 provides communication between computers 31 using objects that are instances of object classes that are defined by attributes of objects, operations on objects and associations between objects. The distributed object-oriented software layer 22 comprises instructions that distribute objects of the same class to computers 31 that are linked by the attributes, operations and associations between the objects within the class in response to the creation of a new object on one of the computers 31. Communication between the computers 31 is provided using predefined, well-known, communication protocols, including asynchronous remote procedure call (ARPC), remote procedure call (RPC), asynchronous message communication (AMC), and broadcast protocols. The distributed object-oriented software layer 22 interfaces with the distributed middle software layer 21 disposed on each computer 31 and causes generation of the intermediate instructions.

Fig. 3 shows that large scale reuse is achieved by systematically supporting variability in an underlying commercial off-the-shelf (COTS) and hardware environment that includes the distributed middleware layer 21 (such as the Universal Network Architecture Services (UNAS) software) available from TRW Data Technologies Division as well as in the application specific functionality, which provides the ability to add new functionality or tailor existing functionality. The separation of the functional concerns from those of building a reliable distributed system is achieved by embodying resources that route data (distribute objects) in the distributed virtual software interface 22 and providing a the simple, powerful interface 22 that permits application development. The distributed virtual software interface 22 has been designed with a number of architectural objectives including portability, support of an object-oriented design, support for the distribution of objects, support for event driven solutions, support for late binding of application processes to the actual physical architecture (virtual node - physical node mapping), and support for fault tolerance.

The distributed virtual software interface 22 is portable because it does not depend on non-portable COTS software products. it uses a well-known International Standards Organization (ISO) POSIX interface and Simple Network Management Protocol (SNMP) standards, uses Ada type/byte stream conversions for communication and storage of data in a heterogeneous environment. and isolates hardware or operating system dependent features to software packages with portable specifications. The distributed virtual machine 22 uses the UNAS COTS software layer 21 which is portable to a wide range of hardware and operating system platforms.

A key aspect of the distributed virtual software interface 22 is its distributed object service which provides access to objects of a given class in a transparent manner with respect to their physical locations. The distributed object service allows distributed application classes to be designed with interfaces where the details of the distributed environment are encapsulated behind the class interface. The distributed object service provides access to objects in a distributed environment. supports physical distribution of object state (through replication) within the distributed environment while ensuring data consistency, and provides class operation delegation which allows a class operation to be invoked in one process and executed in another. such as on a different computer 31 on the network 13.

The distributed virtual machine 22 also supports large scale reuse of the software through its support for event-driven invocation (implicit invocation). Event-driven invocation reduces coupling between cooperating objects and helps build software architectures that are more resilient to changes. The distributed virtual machine 22 supports the addition of new classes and customer specific functionality on top of core application classes via implicit invocation based on changes in the distributed state of the core application classes. This is accomplished through registration, using a notification (NOTI) protocol. that allow classes to "register to be notified" of changes to the distributed application state. Once registered. these classes have their operations implicitly invoked when a specified change in the distributed state occurs. This allows new application classes to be added on top of a core application without modification (or changes in compilation dependencies) to the core application.

The distributed virtual machine 22 permits design decisions as to how distributed applications are ultimately partitioned into independent processes (executing Ada programs) and how those processes are allocated to the available distributed hardware resources to be deferred until late in an application development process and easily changed during the development process. This late binding to the physical architecture and repartitioning into processes (with no or little code modifications) makes systems built on top of the distributed virtual machine 22 easily managed during

development (when small, simplified configurations are often desirable). This late binding aspect of the present invention also makes the systems easily scaleable to widely different hardware configurations in support of large scale software reuse. The distributed virtual machine 22 provides direct support for building fault-tolerant distributed applications. Fault-tolerance is supported through a well-defined fault detection and recovery taxonomy, and through automatic and manual reconfiguration of the processes within a local area based distributed network 13 (LAN) of computers 31.

The architecture of the distributed virtual machine 22 will be further described with reference to Figs. 4-8. The architecture of the distributed virtual machine 22 may be "viewed" from different perspectives including: a static view (Fig. 4) that describes the organization of the software in a development environment (and its compilation dependencies), a logical view (Fig. 5) that is an object oriented model of the design, a dynamic view (Fig. 6) that captures the concurrency and synchronization aspects of the design, a physical view (Fig. 7) that describes the mapping of the software onto the hardware, and a scenario view (Fig. 7) that describes its use. Fig. 8 illustrates elements of a distributed class employed with the distributed virtual machine 22.

Referring to Fig. 4, it shows a static view of a typical software architecture 20 built using the distributed virtual machine 22. The architecture 20 is divided into layers 51-54 where each layer only depends on the layers 51-54 below it, in that all compilation dependencies are downward only). In this view, the distributed virtual machine 22 sits at the bottom of the developed software hierarchy in terms of dependencies. The distributed virtual machine 22 depends on a POSIX interface to the operating system 11, a small number of operating system services outside of the POSIX interface, and the UNAS software layer 21 (Fig. 2). The distributed virtual machine 22 provides a simple, but complete interface to classes of applications 24 that directly make use of it. Exported services provided by the distributed virtual machine 22 (object distribution, time distribution, and synchronization of objects, event and error recording) allow application developers to focus on developing application functionality, with minimal concern regarding how the application operates in a distributed environment. Because the distributed virtual machine 22 supports event driven invocation, a top layer 51 of functionality is supported that makes use of the core application layer 52 and is able to initiate operations on the core application 24 and have its own operations implicitly invoked when changes in the distributed state of the core application 24 occur.

Fig. 5 shows a logical view of classes 61-64 that use distributed object services 60 of the distributed virtual machine 22. The major class 61 in this example is a flight class 61 that provides services to other higher level clients 71-73, which in this example

are an traffic client 71, an airline regulatory client 72, and an airline flight strip client 73. The flight class 61 uses distributed object services 60 to enable its operations and state to be accessed and manipulated by clients 71-73 that are physically distributed. The external interface (to the air traffic and airline regulatory clients 71, 72) and user interface classes 62-64 have a "use" relationship with the flight class 60, allowing the flight class 60 to be independent of its clients 71-73. Even though the class relationship is in this direction, the distributed object services 60 allow the higher level classes 61-64 to have their operations implicitly invoked when client specified changes occur to the state of the flight class 61.

A scenario view is used to show how elements in the static and logical views work together to provide end-to-end (externally visible stimulus and response) operational capabilities. Scenarios are selected to demonstrate the certain "slices" through the architecture 20, showing how various elements interact to achieve a common objective. The design of a scenario is expressed using object scenario diagrams that capture the dynamic semantics and operational relationships between objects.

Because of the architectural framework, and unique interclass relationships that the distributed virtual machine 22 provides to applications built on top of it, a Booch object scenario diagram notation has is used to explicitly make these interactions visible. Classes 61-64 implemented using the distributed object services 60 support the following kinds of interactions with their clients 71-73: operation invocation using normal Ada subprogram call semantics (referred to as "CALL"), operation invocation using synchronous RPC semantics (referred to as "RPC"), operation invocation using asynchronous RPC semantics (referred to as "ARPC"), and operation invocation using implicit notification semantics (referred to as "NOTI").

The annotation of the associated object scenario diagrams with an appropriate operation prefix (CALL, RPC, ARPC, NOTI) has been found to be an effective technique for documenting the intended inter-class design decisions while providing insight into the dynamic architectural view. Fig. 6 illustrates an example object scenario based on the class diagram of Fig. 5, using additional operation annotations.

More specifically, Fig. 6 shows a typical pattern of an object 74 representing the interests of an external system 71 (the air traffic client 71) using ARPC semantics when invoking a file flight plan operation 75 exported by the flight class 61 (Fig. 5). This allows the file flight plan operation 75 to be delegated and performed elsewhere within the distributed system 20. The air traffic object 74 asynchronously receives results from the invoked file flight plan operation 75 (e.g., a semantic response 75a indicating the outcome of the requested operation). The delegation and subsequent

implementation of the file flight plan operation 75 is encapsulated within the flight class 61. As part of the implementation of the file flight plan operation 75, a distributed object 76 is used to commit the newly created object state. This allows the distributed object services 60 to subsequently redistribute (replicate) the object 76 to other processes and computers 31 (or nodes 31), such as an airline regulatory object 77 (the airline regulatory client 72) and electronic flight strip 78 (the electronic flight strip client 73) within the distributed system 20 that require such objects. Finally, any other client classes that have expressed interest in newly created flights flight plans 75 will have their operations implicitly invoked using the notification services (NOTI) provided by the distributed object services 60.

The dynamic and physical views of a system having the distributed virtual machine architecture 20 are shown in Fig. 7. These views show the effect of building distributed application classes using the services of the distributed virtual machine 22. In Fig. 7, the same sequence of operations as shown in the scenario view of Fig. 6 is depicted, but an actual distribution of the resulting software is shown. In this case, the file flight plan operation 75 invoked in a process 81 (Process 1) on one computer 31 (Node A), is implemented in another process 82 (Process 2) on another computer 31 (Node B), and reactions to the newly created object 73 are triggered in different processes 83-85 on various computers 31 (Nodes B-D).

An important property of the distributed object services 60 of the distributed virtual machine 22 is that inter-process communication within the architecture 20 is between instances of the same class. This is because the architecture of the distributed virtual machine 22 allows distributed classes to encapsulate, behind their interfaces, details of any physical distribution of objects that occurs. A client 71-73 (such as the air traffic client 71, the airline regulatory client 72, or the electronic flight strip client 73) or client class simply invokes a particular class operation using either synchronous or asynchronous semantics (this is a client decision), while the details of how and where the operation is actually executed is not visible to the client 71-73.

Fig. 7 also shows the client/server relationships that exist between processes 81-85 and classes 61-64. The relationship between two processes 81-85 is always client/server, and a single process 81-85 can be both a client and a server. In Fig. 7, the first process 81 (Process 1) is a client of the second process 82 (Process 2), and the second process 82 (Process 2) is a client of the third process 83 (Process 3). The client/server model also extends to the class level. In addition, in order to describe a distributed class, the a service interface is used (in addition to client and server). The air traffic class 62 and the flight class 61 have a client/server relationship because the air traffic class 62 invokes operations of the flight class 61. However, the flight class 61

operates as a service interface and a server depending on which process 82, 83 (Process 2 or 3) it executes. The flight class 61 in the first process 81 (Process 1) operates as a service interface because it is primarily responsible for delegating operations to the instance of the class operating as a server in the second process 82 (Process 2). This client/service-interface/server relationship is applied uniformly in all distributed class interactions.

Fig. 7 shows the role of the distributed object service 60 (Fig. 5) in physically distributing copies of objects 76 to nodes 31 (computers 31) and processes 82-85 where they are required. The distributed object 76 in the third process 83 (Process 3) is a reference copy of the object 73, while distributed objects 76 in the second, fourth and fifth processes 82, 84, 85 (Processes 2, 4, 5) are surrogate copies of the object 76. It is the responsibility of the distributed object service 60 to ensure all objects 76 remain consistent and provide uniform access to object state independent of the existence of a local surrogate copy.

The ability to provide simple exported interfaces for distributed classes 61-64 that are immune to changes in the dynamic and physical architectures, and the operations provided by the distributed virtual machine 22, allow details of the distribution to change without any impact on the client or server classes 61-64. Each independent process 81-85 that is based on services provided by the distributed virtual machine 22 conceptually has a single thread of control. Processing is always initiated by receipt of a message (ARPC, RPC, NOTI) and may occur asynchronously or periodically. This simplified concurrency model within processes 81-85 greatly reduces debugging complexity that is often found in multi-tasking applications and eliminates any concern for mutual exclusion within the context of a process 81-85. Where additional asynchronous behavior is required, the distributed virtual machine 22 provides a light weight thread scheduler (referred to as the Pivot thread scheduler) whose services are detailed below.

A more detailed description of the services provided by the distributed virtual machine 22 are described below in a bottom-up fashion, starting with low-level services and then higher level distributed services. The distributed virtual machine 22 provides a number of low level services for its own use as well as for applications. The distributed virtual machine 22 uses subprogram variables. In order to provide a functionality similar to types "access to subprograms" in Ada 95, a subprogram variable service allows the creation of objects that are "pointers" to procedures with specific parameter profiles. Subprogram variables are "type safe", and are modeled after Ada 95 access to subprograms. The use of subprogram variables in the distributed virtual machine 22 provides support for event-driven (implicit) invocation.

An off-line tool, ATIG (Ada Type Interchange Generator, developed by Little Tree Consulting), is used to automatically generate Ada code that converts Ada objects into common "byte stream" data types and vice-versa. This mechanism is similar to the stream I/O attributes of Ada 95, but is implemented in Ada 83/87. This tool is widely used in the implementation of the distributed virtual machine. It allows heterogeneous communication between processes with no need for representation clauses. Two formats are supported, including a binary format and a human-legible (text) format.

The Pivot thread scheduler is a very light weight asynchronous thread scheduler that does not use Ada tasks. Services provided by the Pivot thread scheduler have been developed to provide a high-performance scheduler where a potentially large number of dynamically created asynchronous threads are required. High performance is obtained because there are no task or process context switches, nor any need for additional mutual exclusion mechanisms. A Pivot thread is associated with an Ada procedure. Each Pivot thread is a member of a "group" where all threads of the same group share a common context. Each thread can also have a private context, persistent between invocations of the thread.

There are different kinds of threads and ways to invoke them. A thread may be time sensitive (periodic), event sensitive (reacting to the occurrence of a global stimulus), command sensitive (reacting to a stimulus directed to its group), response sensitive (reacting to a stimulus specifically directed to the thread) or synchronous (directly invoked). The Pivot thread scheduler also supports implicit invocation. It provides services of a bookkeeper, a dispatcher, and a watchdog. It manages service requests and delivers subsequent responses. Such schedulers are generally well understood by Ada programmers.

The Pivot thread scheduler is a software component that provides threads of control. The Pivot scheduler provides services of a bookkeeper, maintaining knowledge of which threads have expressed an interest in particular events. The Pivot thread scheduler provides services of a dispatcher, invoking threads in response to the occurrence of events that have been designated as "interesting". The Pivot thread scheduler provides services of a broker, managing a context that may be shared by a number of threads that contribute to a collective goal. The Pivot thread scheduler provides services of a watchdog, alerting those threads that have requested to be invoked when a specified period of time has elapsed. The Pivot thread scheduler provides a mechanism that orchestrates execution of an application whose behavior is highly dependent upon a number of dynamic state variables.

Threads are references to subprograms whose invocations by the thread scheduler are based on three different sensitivities: events, responses, and times. There

are three categories of Pivot threads that exploit these sensitivities, namely, event sensitive, response sensitive, and time sensitive threads, respectively.

An application creates an event sensitive thread that is invoked as the result of specific events, which, for example, may be produced in response to distributed object notifications. When an application "produces" an event, all threads having expressed an interest in the event are "dispatched" by the Pivot thread scheduler. Whereas the creation of an event results in the dispatch of any number of event sensitive threads that have previously expressed interest, a response sensitive thread is created to direct data to a single thread. The application creates a time sensitive thread that is invoked as the result of an expiry condition specified in an argument list. In general, the creation of threads using the Pivot thread scheduler is well understood by Ada programmers.

The distributed virtual machine 22 provides several parameterization services to its clients 61-64. The distributed virtual machine 22 uses these parameterization services to provide portability and scalability. These services allow each executable program instance to have its behavior tailored by a set of resource data. Thus, a program instance is a pair: [executable binary, resources]. A resource is an initial value that was not written explicitly in the source code but must be provided for the program to run. A resource can be as simple as a single integer value or as complex as a table of arbitrary values. Two kinds of resources include constant and variable system parameters which allow the parameterization of applications and of the distributed virtual machine 22 before start of execution (constant system parameters) and during runtime (variable system parameters). Constant system parameters may be used during elaboration (for example, for the bounds of subtypes). Services to define and manage collections of system parameters are provided by the distributed virtual machine 22.

The system parameter mechanism allows generation of binary data files that are read by the software when it is initially loaded. The advantage of this mechanism is that it allows modification of these parameters, thereby modifying the software behavior in some desired way without the requirement to modify source code and recompile and relink all of the software. This significantly shortens the time it takes to make certain changes to the system.

The distributed virtual machine 22 permits the use of Ada exceptions. However, an application running under normal operating conditions should not raise an Ada exception. The explicit raising of an exception under abnormal conditions is always made through an error reporting mechanism, by reporting an error with an appropriate severity. The different severities include informative, which is used to report interesting, but infrequent events; warning, which is used to report an unexpected condition within the software, but local recovery was possible and the

software is able to continue to provide service: serious but not fatal, which is used to report that the current operation is unable to be completed, but there is no reason that subsequent operations won't be successful; controlled and fatal, which is used to report the occurrence of a fatal event, but the software knows what's wrong and is confident that a controlled shutdown of the process will be successful; and wild and fatal, which is used to report conditions that should never happen, such as if there is a strong indication of a programming error, or the loss of an essential system resource (memory). Only the last three reports result in an exception, and only the exception raised for severity "serious but not fatal" is handled by the general application code. The primary motivation for this strategy is to avoid exception handling code from creeping all over the place and dwarfing the useful code, a phenomenon often observed in large Ada systems. The error reporting mechanism allows applications to define and report their own errors and to parameterize the error text through resource data. Reported errors are centralized for recording, for display and for subsequent query.

The distributed virtual machine 22 provides services that allow an application to be instrumented for the collection of performance data. The distributed virtual machine uses this capability for some of its operations in implementing the distributed services. These services support the grouping of measurements for input to statistical analysis and threshold evaluation. The services support the development of higher level responses (i.e., application specific) to sustained poor performance trends. (NOTE: if this topic is not pertinent to the invention, then it should be deleted. Otherwise, this topic needs to be explained in more detail.)

The distributed virtual machine 22 provides services that allow application classes to achieve intra-class message-based communication independent of their physical location in the architecture. Class communication services are the foundation for all inter-process communication. A class 61-64 may have different roles with regard to inter-process communication: service interface, server or both.

There are four types of communication protocols used in the distributed virtual machine 22. An ARPC protocol is used by the vast majority of classes. This is the standard protocol used for all delegated operations. The caller's thread of control is released immediately after sending the message, and the caller will be notified (asynchronously) when a return message is received. The ARPC model is based on the metaphor of taking one's car in to be serviced at an auto repair shop. The general sequence of events is illustrated as follows.

A client takes a car to the repair shop requesting work to be performed (invokes a class operation). At the same time, the client also provides a phone number (in addition to other normal input parameters) so that the repair shop can call back when the

service is complete (implicitly invoke the client to inform of operation completion). As a result of the request for service, the client is provided a work order that uniquely identifies this service request (an output parameter of the operation request). At some time later, the client (asynchronously) receives a phone call from the repair shop indicating that the requested service (operation) has been completed. Once again, the work order is used to uniquely identify the requested operation. At some time convenient to the client, the client returns to the repair shop (invokes a class operation to retrieve the results of the requested operation). Based on the work order the results of the service (operation) are returned to the client.

An RPC protocol is used by classes that need to implement synchronous communication. The caller's thread of control is blocked until a return message is received or the RPC times out. An AMC protocol is used by classes implementing some ad-hoc communication protocol. A broadcast protocol is used to distribute messages to multiple receivers. The broadcast protocol enables a class to broadcast a message to multiple destinations by sending only one physical message over the network. Class instances can act as transmitters or can subscribe to the broadcast service, in which case they will receive all subsequent broadcast messages of the class.

The distributed object service 60 allows applications to access objects of a class without concern for where the objects are actually located within the distributed system 20. It provide its users with features including uniform access, an application-oriented cache, data consistency, notification, failure recovery, time management, notices, recording services, data extraction services, and tactical configuration. Uniform access refers to the ability of a distributed class (and consequently clients of that class) to access any of its objects without knowledge of the actual location of the object.

With regard to the application-oriented cache (illustrated in Fig. 8), local caching of objects (i.e., the physical distribution of objects values throughout the system) allows distributed classes (and their clients) to make the best possible use of limited memory resources (and network bandwidth) by indicating, on the basis of local knowledge of an object's criteria of interest, that certain objects should be immediately available for read access at any time. This service is known as subscription and the local copies of object values are referred to as surrogates.

A subscription to objects can be expressed either through a filter or by explicit specification of an object's handle. A filter provides a means of specifying a subset of the objects of a class by applying a Boolean predicate to each object of the class when it is updated for those objects that are of interest. The subscription capability directly supports the software design principle of bringing objects to where they are needed rather than always deferring to some central processing resource. Not only does this

provide for improved response times for read access to objects, but it also supports continued access to objects even during failure of other processing elements, thus contributing to the fault tolerance characteristics of the distributed virtual machine.

Data consistency is ensured between multiple copies of a given object within a class (and across different classes). Transactions allow several objects (possibly of different classes) to be accessed automatically, in the sense that modifications performed in the course of the transaction are committed on an all-or-nothing basis. The notification capability allows clients to register to be notified (via a callback) of changes to objects of a distributed class. This capability allows for the reversal of dependency between two classes, and allows the distributed class to only be concerned with maintaining its own state, rather than also concerning itself with the other classes that are interested in those state changes.

The notification capability allows the client class to register interest in objects based on their state or based on changes to their state. As with the subscription capability, a filter may be specified to indicate which objects are of interest. When an object is committed at its central object store, its state is evaluated against the notification filter and if it passes, then the client is notified. In addition to expressing interest in objects via a filter, clients may also express interest based on change to the state of an object. A mutation is a description of the difference between two objects. When two objects are successive versions of the same object, a mutation describes the set of changes that occurred from one version to the next. Mutations allow clients to be notified upon arbitrary change to some set of attributes of an object. The mutation is always used in conjunction with a notification filter and acts as a second level criteria that must be passed before a notification will occur. The mutation of an object is computed in a central object store (at transaction commit time) and distributed as an additional attribute of the object. This allows the mutation to be accessed as any other attribute of the object, allowing clients of the distributed class to base their processing on knowing exactly which attributes changed in the current version of the object.

Automatic failure recovery is supported by keeping one or several backup copies of each object on various processing nodes 31, and ensuring that these copies remain synchronized and consistent.

With regards to time management, the distributed virtual machine 22 simultaneously supports concepts of real and logical time. Logical time is a time that has a linear relation with real time. The parameters of this linear relation may be modified on line, within limits. For each type of time (real and logical) there is a corresponding type of duration (real and logical). In an operational system, logical time

is equivalent to real time. For training or simulation, logical time may be different from real time: it may flow "faster" or "slower", or be "frozen".

The real time clock is provided by the host processor system time. The distributed virtual machine 22 assumes that the different processor system clocks are synchronized by an external mechanism, such as the well-known Network Time Protocol (NTP). The services provided on times and durations are reading the current (real or logical) time, arithmetic on times and durations, and control of the distributed logical time (setting the initial time, modifying the time flow, that is the "speed" of the logical time versus the real time) for all the processes of an application, real and logical timers, "one shot" or cyclic, that invoke a specified operation when they expire.

The distributed virtual machine 22 provides a notice mechanism that is used to disseminate operational events. This is another mechanism that supports the concept of implicit invocation: when a notice is created, it can be received by any interested client in the system without the creator's knowledge of the particular recipients. The notice mechanism is based on a metaphor of pinning notes on a bulletin board and allowing unknown interested parties to read and react to them.

Notices are divided into notice categories, and each notice category is further subdivided into keyword subsets. A notice belongs to a single notice category and is a member of one or more keyword subsets. Each notice contains some information, the type of which depends on the notice category.

A bulletin is a set of notice categories. Each notice category is a member of one or more bulletins. A notice is "posted" on all of the bulletins associated with its category. A notice has one creator and zero or more subscribers. Subscribers may subscribe to individual notice categories, as well as to bulletins. Each notice has a state during its lifetime: created, completed or timed-out.

Notices have several characteristics, in that their lifetimes may be limited, and upon expiration, a notice will be automatically deleted. Subscribers may "respond" to a notice, giving feed-back to the creator. A number of responses to a notice may be required, and when this number of responses is given, the state of the notice is changed to "completed". For notices that need one or more responses, a "response" period may be limited. If at the end of this period, the proper number of responses have not been received, the state of the notice becomes "timed-out". Notices, when they are "timed-out", may be escalated: another notice of a different category, displayed on different bulletins, is created and distributed to potentially different subscribers. A notice may be "updated" by its creator after its creation. Subscribers to a specific notice category may specify the states and keywords for which they want to be notified.

The distributed virtual machine 22 provides a recording mechanism that allows applications to record on-line distributed operational data in a central location. The metaphor of a sound track studio is used to implement this mechanism, with classes such as signal, mixer, recorder, tape and tape library. Similar recorded data of all processes in an application are centralized on one physical location. Fault tolerance mechanisms are used to ensure continuous operation of the recording services. Data extraction services are also provided to support application level data reduction and report generation.

Tactical configuration services of the distributed virtual machine 22 provide capabilities to manage and control mapping of the dynamic architecture to the physical architecture. This is accomplished through real-time monitoring of the current system configuration, manually controlling the current system configuration (startup, shut-down, reconfiguration), and automatic reconfiguration in response to system faults.

The distributed virtual machine concept for management of a network starts with the notion of a logical network, where a logical network is a set of communicating processes, operating within a common (logical) time base, that provide a service. The distributed virtual machine 22 supports managing multiple simultaneous logical networks that share the same set of physical resources (nodes 31 and physical network 13). Each logical network includes many settings where each setting is a named collection of processes that may execute on a node 31 of the physical network 13.

The monitoring capabilities provide insight into the current logical networks, settings and processes as well as insight into the current allocation of nodes 31 to the physical network 13. The manual reconfiguration services allow for controlled system level startup and shutdown as well as the execution of scripted reconfigurations without service interruption. Common reconfiguration actions can be saved as a script and recalled at any time.

A description of how the distributed virtual machine 22 may be used to build fault tolerant distributed applications is described below. Fault tolerance is defined and a general taxonomy of faults that the software can detect and respond to is discussed. Each of the software services provided by distributed virtual machine 22 that play a significant role in the fault tolerance approach is discussed.

A distributed application provides services. A service is correct if, in response to inputs, it behaves in a manner that is consistent with its specification. When a service fails to behave in its prescribed manner, then a fault has occurred. If a fault is unhandled or exists for a prolonged time, then a failure has occurred. The fault tolerant mechanisms of the distributed virtual machine 22 prevents failures from occurring.

From the very specific to the very general, software faults can be classified into the following hierarchy which includes crash fault, omission fault, timing fault and arbitrary fault. A crash fault is the failure of a service to respond to an input and all subsequent inputs. An omission fault is a super-set of crash faults, and includes the case where a service fails to respond to a particular input, but may respond to subsequent inputs. A timing fault occurs when a service fails to respond or responds too early or too late. Performance faults are a subset of timing faults that are predominantly due to responses that are too late. An arbitrary fault is either a timing fault, or the service produces a response different from the one specified.

The distributed virtual machine 22 provides a variety of integrated services to detect and recover from the various classes of software faults. Tactical configuration services are responsible for detection and process level recovery from crash faults. Class communication services are responsible for the masking (from the application classes) of crash faults and the detection of omission faults. The distributed object services are responsible for ensuring data consistency and continued access after the recovery from crash and omission faults. Performance measurement services are responsible for the detection of performance faults.

The tactical configuration services that support the detection and recovery from crash faults are described below. The crash faults handled by the tactical configuration services can be further divided into process faults and node faults. Process faults can, in turn, be divided into three sub-categories depending on the kind of failures they may lead to: blindfolded, wild, and controlled failures.

Blindfolded failures occur when a process fails without even being aware of it (for example, a class enters an infinite loop). The watchdog capabilities of the UNAS software layer 21 are responsible for detecting these failures. Wild failures are detected from within the process, but their origin or impact is unknown. These failures manifest themselves as unhandled exceptions and are detected by an exception handler at the bottom stack frame of the main task in each process. Controlled failures are the result of an explicit action by the software that has detected a fatal condition. Controlled failures manifest themselves as a unique Ada exception raised in response to the reporting of a fatal condition. A controlled failure is different from a wild failure in the sense that with a controlled failure, the process can be shutdown in an organized fashion, giving the applications an opportunity to save data or do any other appropriate task.

To enable tailoring of recovery processing in response crash faults, each active process operates in a process mode, which helps define the process's fault tolerance strategy. The distributed virtual machine 22 supports the following predefined process

modes: solitary, sharing, primary, and secondary. A process is in the solitary mode if it is actively providing services to client processes, and there are no associated sharing or secondary processes active. A process is in the sharing mode if it is actively providing services while operating in a network configuration where there are multiple instances of this same process actively sharing the service load. A process is in the primary mode if it is actively providing services and is operating in a network configuration where it has been paired with another process in secondary mode.

A process in the secondary mode is not actively providing services (to client processes), but is potentially interacting with another process instance operating in primary mode. The secondary mode process is responsible for being prepared to take over the services provided by the primary mode process in the event of a fault.

Resource data, along with the mode of the process experiencing the failure, determine the particular recovery action to be taken in response to a crash fault. Recovery actions can be specified for individual processes, for individual nodes 31, or for the entire network, providing a natural hierarchy of possible recovery actions dependent on the type of failure (process or node 31) and the current system configuration. In the case of an entire node failure, a special mode (node failure) is used to direct the particular recovery actions.

Tactical configuration services support recovery strategies in response to crash faults including starting a new process on the same node 31 or on a different node 31; changing the mode of an existing process on the same node 31 or a different node 31; shutting down an existing process on the same node 31 or a different node 31; and performing a soft shutdown and warm restart of the failed process (applicable for controlled failures).

The class communication services play two roles in the fault tolerance design: aiding in the masking of crash faults by supporting automatic re-establishment of client/server communication paths as part of node 31 or process recovery actions taken by the tactical configuration services, and providing automatic detection of omission faults.

The masking of crash faults from the majority of the application code occurs as the result of the automatic recovery actions described in the previous section as well as the automatic re-establishment of client/server communication paths performed by the class communication services. After the recovery of a failed server process (on the same or different node 31), the class communication services automatically determine the new address for the server and re-establish the communication connections between that server and all of its clients.

In addition, the class communication services export operations that allow distributed classes to be built such that they take explicit advantage of operating in the sharing or primary/secondary configuration. These operations support the design of higher level fault recovery strategies within distributed classes by allowing them to naturally react to changes in the network topology.

Omission faults are a super-set of crash faults and include the case where a server fails to respond to a request but has not crashed. Omission fault detection has been built into the asynchronous and synchronous remote procedure call (ARPC and RPC) protocols provided by the class communication services through the automatic detection of time-out conditions. These protocols detect the failure of a server to respond to a client request within a specified duration. If a client does not receive a response within the specified duration, an error response is generated and returned to the client indicating that the request has not completed due to a time-out. The RPC and ARPC protocols guarantee that once a client receives a time-out indication, the client will not receive the real response later, even if it actually arrives.

The time-outs built into the ARPC and RPC protocols are only intended to detect conditions when something has really gone wrong, rather than detecting the condition where responses are just late (timing fault). Consequently, the time-out duration associated with an ARPC or RPC is several times larger than the actual time the request is expected to take.

There is a tension between the goals of data integrity, fast response time and fault-tolerance. Providing data integrity requires that only a single copy of each object be maintained, and all accesses to an object be serialized. Fast response time (less than a second or even half a second) to air traffic controllers, and high fault-tolerance and availability of the system all require to bringing objects close to where they are being used. This, in turn, leads to a high degree of redundant information. The distributed object services resolve this tension by providing fast read access to objects through replication, while still ensuring data integrity, even in the presence of faults.

Fig. 8 illustrates independent processing elements or components that comprise a class implemented using distributed object services of the distributed virtual machine 22. The components include class instances providing service interface behavior, class instances providing server role behavior, and central object storage/data consistency/availability assurance.

Referring to Fig. 8, a plurality of individual computers 31 are shown that each run processes 81-85 (i.e., Ada programs). The processes 81, 82 shown in the top portion of Fig. 8 each contain a client application 92a, 92b (Client) making use of the services of a distributed class 91. The distributed class 91 illustrated on the left side of

Fig. 8 is in a service interface role and communicates with one or more instances in a server role (shown by the processes on the right side of Fig. 8). The bi-directional arrow connecting the computers 31 represents internal communication that occurs during operation delegation. Each process 81, 82 that contains a distributed class 91 also contains a single transaction manager 94. Both the distributed class 91 and the transaction manager 94 communicate with the primary central object store 95 associated with the distributed class 91. The central object store 95 is responsible for committing transactions, distributing surrogate copies of objects to proper remote caches 96 , and communicating with a secondary central object store 97.

Of these core capabilities of the distributed object services, the data consistency, notification, and automatic failure recovery capabilities play key roles in the fault tolerance system 20. Ensuring data consistency in the presence of crash faults is the combined responsibility of the transaction manager 94 and the central object store 95 provided by the distributed object services. The transaction manager 94 provides operations to initiate, roll back, and commit transactions. Integrity is ensured by keeping in a transaction log all modifications to objects requested as part of that transaction, and making their effect permanent only upon the successful completion of the commit operation. If the transaction is rolled back, or is never committed due to the occurrence of a fault, the transaction log is discarded and no change to the object state occurs.

The design of the transaction manager 94 was heavily influenced by the desire for it to be as "light-weight" as possible. This is why the transaction manager 94 is not based on the use of locks or two-phase-commit protocols, but an approach of detecting object version mismatches at transaction commit time (rather than trying to secure locks on objects at each step of the transaction).

From the above description of the transaction capabilities, the transaction manager 94 ensures data integrity in the case of a crash fault occurring in the committing process (prior to the transaction being committed). The transaction manager 94 only partially accounts for ensuring data integrity in the case of a crash fault occurring in the primary central object store 95. The automatic failure recovery capability provided by the distributed object services provides the balance of the capabilities. The automatic failure recovery capabilities provided by the distributed object services provide tolerance in the presence of crash faults occurring in the primary central object store 95 or in the case of complete node failures. The automatic failure recovery functionality is an example of the primary/secondary process mode capabilities described above.

For each central object store operating in the primary process mode, there may exist another central object store operating in the secondary process mode on a different computer 31. It is the responsibility of the primary central object store 95 to keep the

secondary central object store 97 synchronized regarding the state of all objects. The communication that occurs when committing a transaction, described above, as well as the communication that occurs when initializing the secondary central object store, accomplishes this synchronization.

In the case where the tactical configuration mechanism detects a crash fault in the primary central object store process, the automatically initiated recovery action causes the secondary central object store process to undergo a mode change, from secondary to primary. In addition, it is the responsibility of the central object store to provide persistence for all objects. This persistency is relied upon when the system is initially started or re-started from a cold state.

The notification capability provided by the distributed object services provides a powerful form of communication between classes by allowing an operation of one class to be implicitly invoked as the result of a change to the state of an object in another class. Since notifications trigger second-order processing on other classes, it is important that these events are not lost in the presence of faults. Performing the evaluation of notification criteria (and subsequent notifications), not only at transaction commit time, but also when the notification criteria are initially registered or re-registered, ensures notification events occur, even in the presence of faults. By evaluating notification criteria at the time of registration, the distributed virtual machine 22 can recover from both crash faults in the primary central object store 95, and in any second-order processing (i.e., the second-order processing will occur).

The distributed virtual machine 22 has been reduced to practice and is currently undergoing testing. The entire framework currently represents over 50,000 source lines of Ada code (not including the UNAS software layer 21). The distributed virtual machine 22 has been tested in distributed systems that have physical networks containing over fifty computers 31 or nodes 31.

Thus, a distributed virtual software interface that interfaces between computers, operating systems and applications that run on the computers of a distributed, multi-computer fault tolerant computer system has been disclosed. It is to be understood that the described embodiment is merely illustrative of some of the many specific embodiments which represent applications of the principles of the present invention. Clearly, numerous and other arrangements can be readily devised by those skilled in the art without departing from the scope of the invention.

CLAIMS

What is claimed is:

1. An object-oriented software architecture (20) implemented in a distributed computer system (30) having a plurality of computers (31) that are interconnected by way of a network (13), and wherein each computer (31) comprises an operating system, said architecture (20) characterized by:

a distributed intermediate software layer (21) that is distributed among the plurality of computers (31) that interfaces with the operating system (11) of the computer (31) on which it is disposed and that generates intermediate instructions that cause the operating system (11) to implement primitive operating system instructions in response thereto;

a distributed object-oriented software layer (22) distributed among the plurality of computers (31) that provides communication between computers (31) using objects that are instances of object classes that are defined by attributes of objects, operations on objects and associations between objects, and wherein the distributed object-oriented software layer (22) comprises instructions that distribute objects of the same class to computers (31) that are linked by the attributes, operations and associations between the objects within the class in response to the creation of a new object on one of the computers (31), and wherein communication between computers (31) is provided using predefined communication protocols, and wherein the distributed object-oriented software layer (22) interfaces with the distributed middle software layer (21) disposed on each respective computer (31) and causes generation of the intermediate instructions; and

at least one software application disposed on each computer (31) that interfaces with the distributed object-oriented software layer (22) disposed on the respective computer (31) and processes objects distributed to it.

2. The architecture (20) of Claim 1 wherein the operating system (11) is characterized by a UNIX operating system (11).

3. The architecture (20) of Claim 3 wherein the distributed intermediate software layer (21) is characterized by Universal Network Architecture Services software.

4. The architecture (20) of Claim 3 wherein the distributed object-oriented software layer (22) supports multiple communications protocols.

5. The architecture (20) of Claim 1 wherein the distributed object-oriented software layer (22) supports an asynchronous remote procedure call communications protocol.

6. The architecture (20) of Claim 5 wherein the distributed object-oriented software layer (22) supports a remote procedure call communications protocol.

7. The architecture (20) of Claim 5 wherein the distributed object-oriented software layer (22) supports an asynchronous message call communications protocol.

8. The architecture (20) of Claim 5 wherein the distributed object-oriented software layer (22) supports a broadcast communications protocol.

9. The architecture (20) of Claim 5 wherein the distributed object-oriented software layer (22) provides for class-based communications between respective ones of the computers (31).

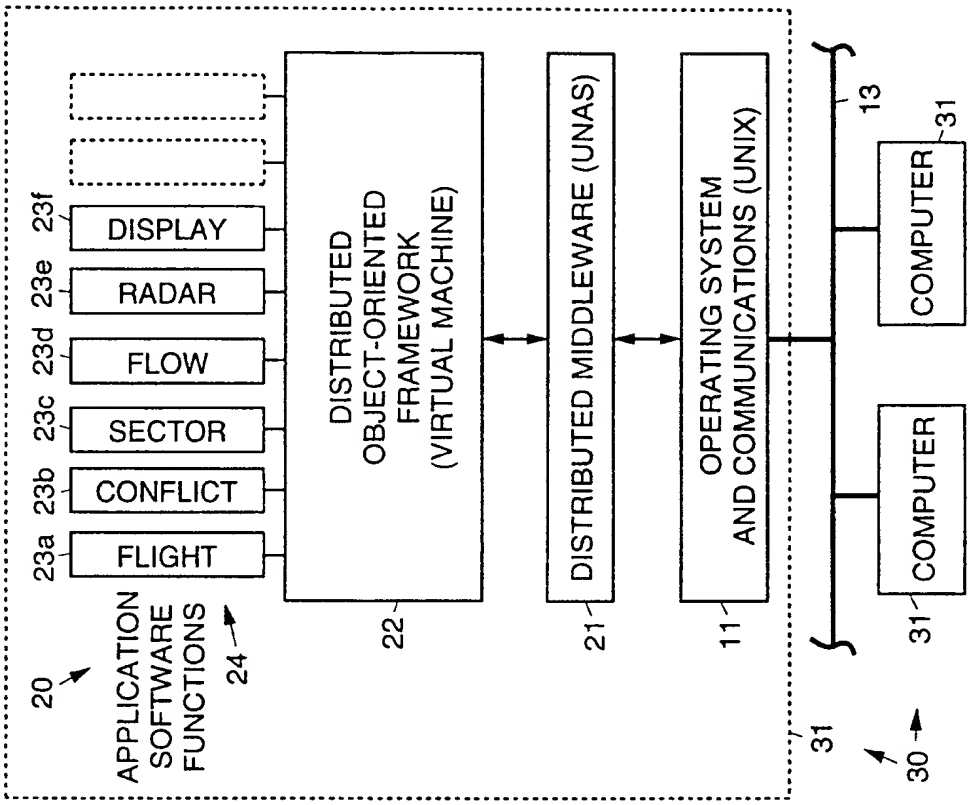


FIG. 2

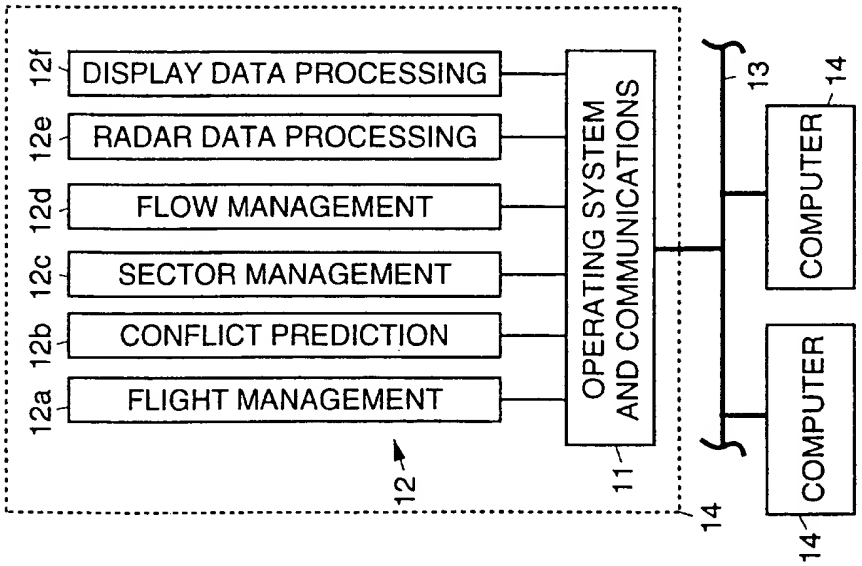


FIG. 1
(PRIOR ART)

10

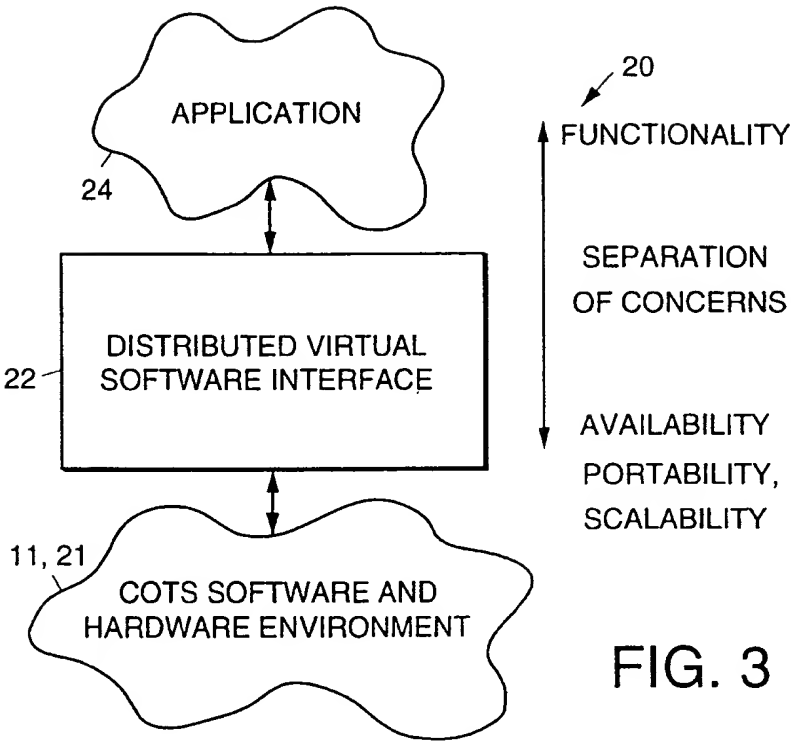
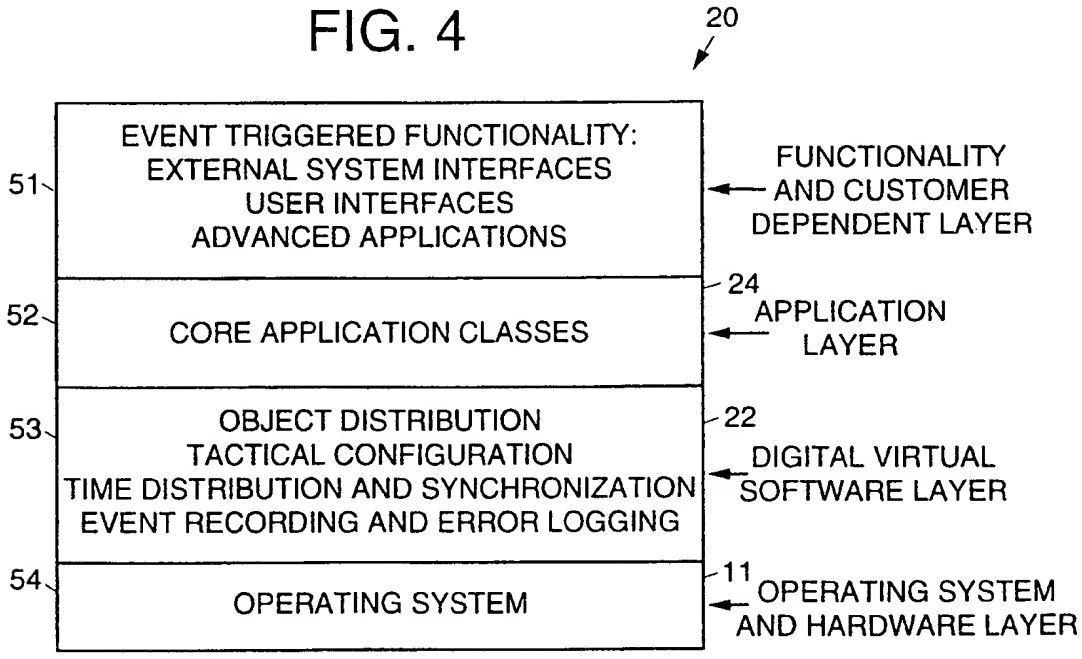

APPLICATION
24
20
FUNCTIONALITY
SEPARATION
OF CONCERNS
DISTRIBUTED VIRTUAL
SOFTWARE INTERFACE
22
AVAILABILITY
PORTABILITY,
SCALABILITY
11, 21
COTS SOFTWARE AND
HARDWARE ENVIRONMENT

FIG. 3

FIG. 4


20
EVENT TRIGGERED FUNCTIONALITY:
EXTERNAL SYSTEM INTERFACES
USER INTERFACES
ADVANCED APPLICATIONS
51
FUNCTIONALITY
AND CUSTOMER
DEPENDENT LAYER
CORE APPLICATION CLASSES
52
24
APPLICATION
LAYER
OBJECT DISTRIBUTION
TACTICAL CONFIGURATION
TIME DISTRIBUTION AND SYNCHRONIZATION
EVENT RECORDING AND ERROR LOGGING
53
22
DIGITAL VIRTUAL
SOFTWARE LAYER
OPERATING SYSTEM
54
11
OPERATING SYSTEM
AND HARDWARE LAYER

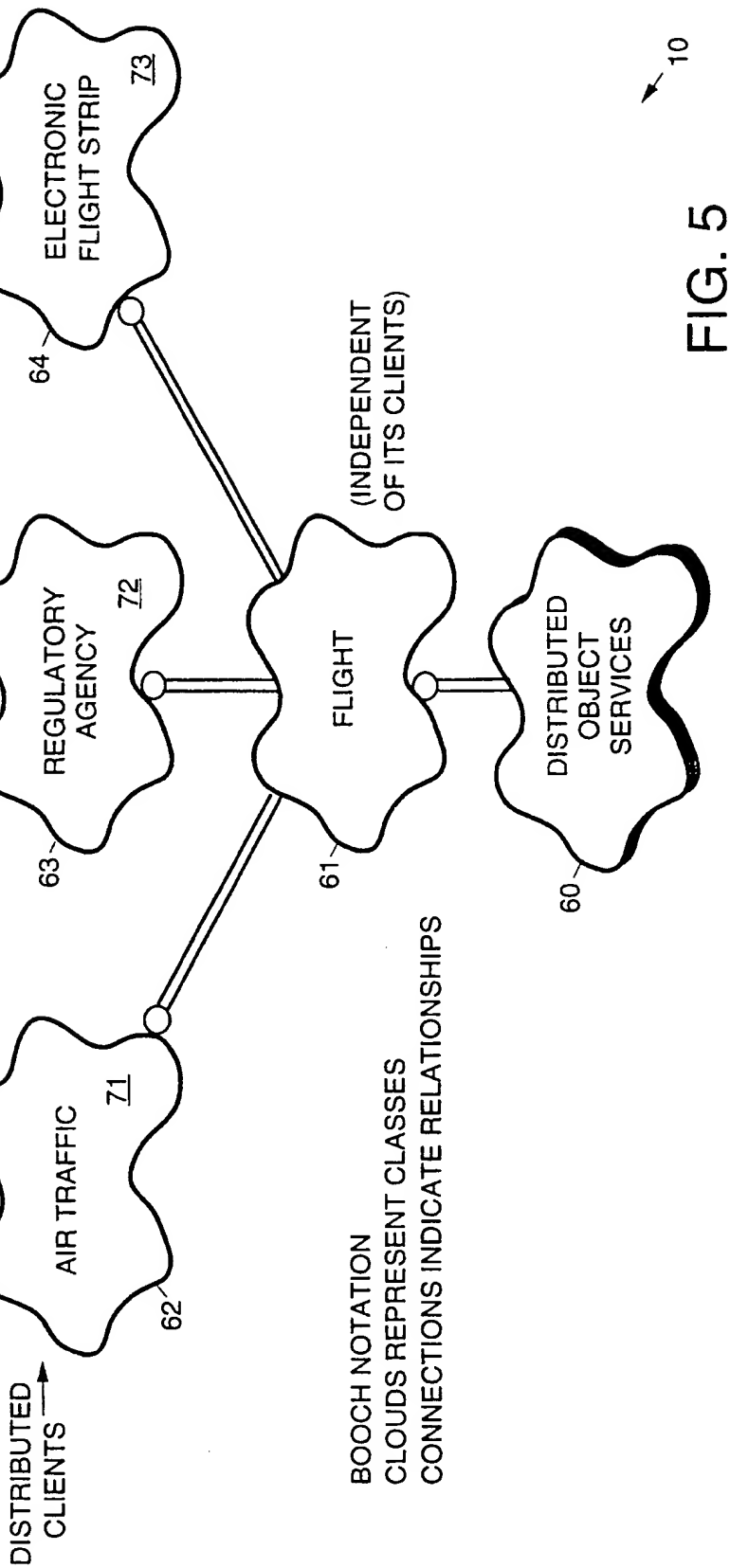

10
ELECTRONIC FLIGHT STRIP
73
64
REGULATORY AGENCY
72
63
FLIGHT
(INDEPENDENT OF ITS CLIENTS)
61
DISTRIBUTED OBJECT SERVICES
60
AIR TRAFFIC
71
62
DISTRIBUTED CLIENTS
BOOCH NOTATION
CLOUDS REPRESENT CLASSES
CONNECTIONS INDICATE RELATIONSHIPS

FIG. 5

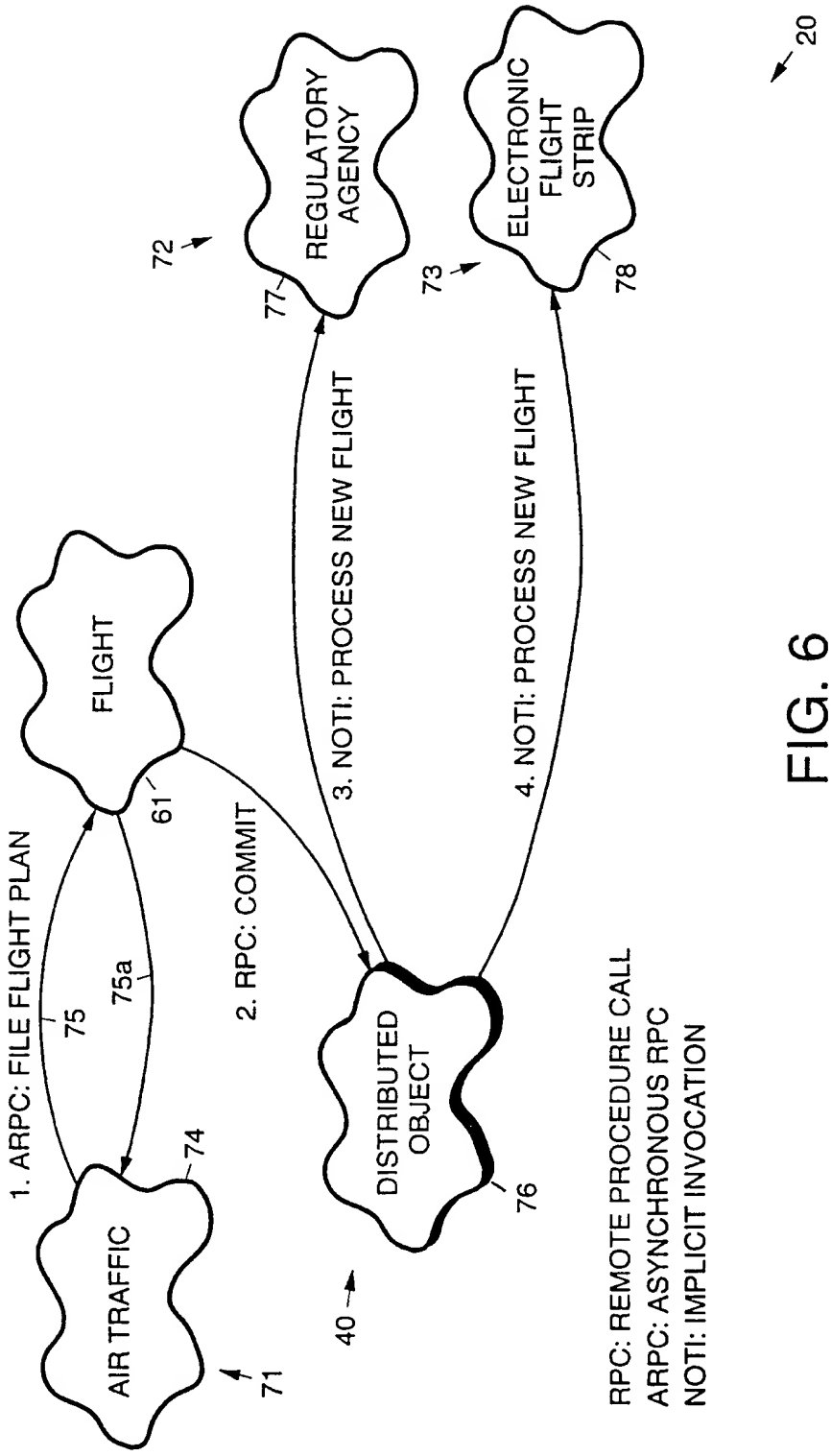

20
72
REGULATORY AGENCY
77
73
ELECTRONIC FLIGHT STRIP
78
FLIGHT
61
1. ARPC: FILE FLIGHT PLAN
75
75a
2. RPC: COMMIT
3. NOTI: PROCESS NEW FLIGHT
4. NOTI: PROCESS NEW FLIGHT
AIR TRAFFIC
74
71
DISTRIBUTED OBJECT
76
40
RPC: REMOTE PROCEDURE CALL
ARPC: ASYNCHRONOUS RPC
NOTI: IMPLICIT INVOCATION

FIG. 6

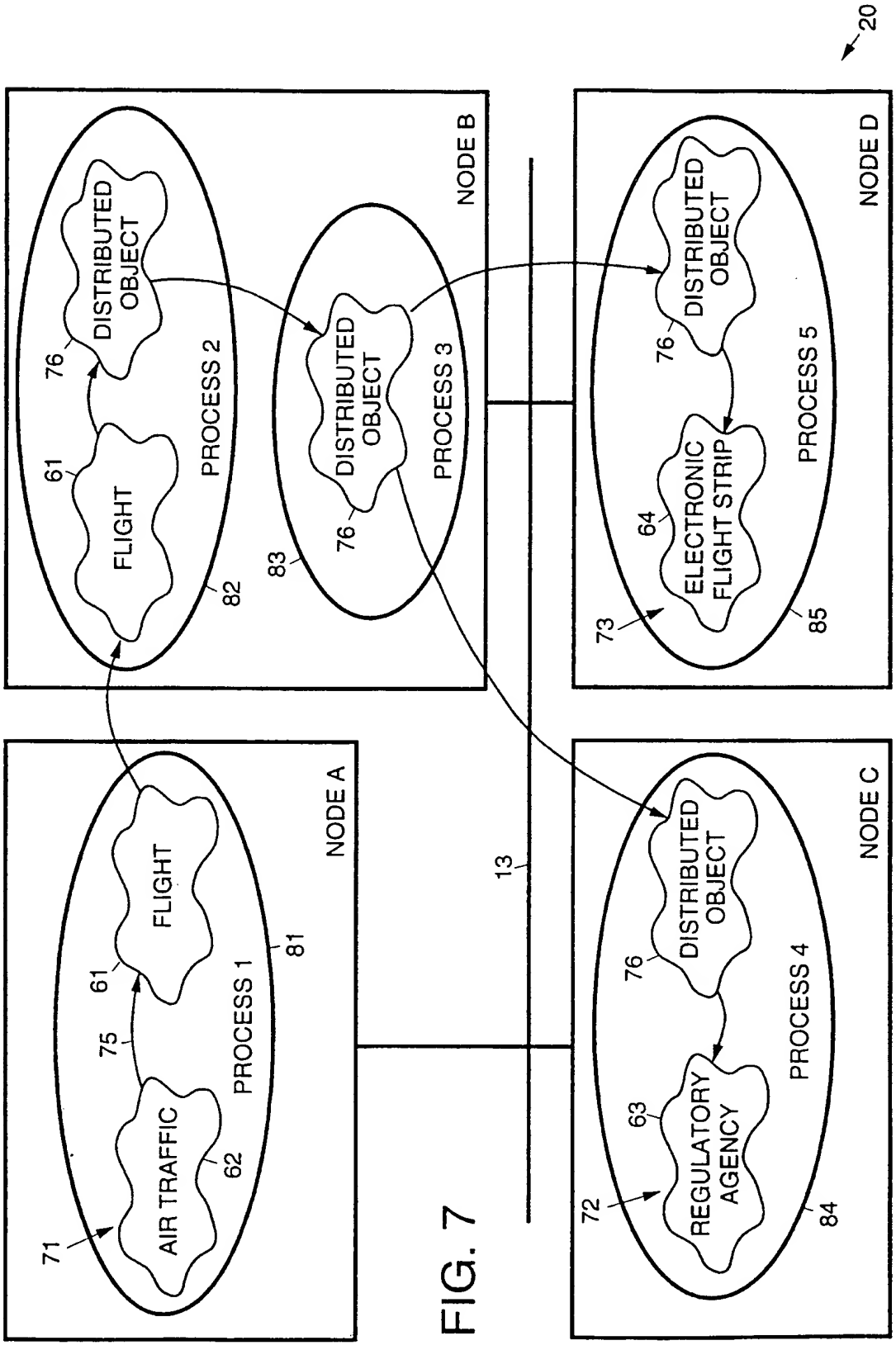

20
NODE A
NODE B
NODE C
NODE D
PROCESS 1
PROCESS 2
PROCESS 3
PROCESS 4
PROCESS 5
AIR TRAFFIC
FLIGHT
FLIGHT
DISTRIBUTED OBJECT
DISTRIBUTED OBJECT
DISTRIBUTED OBJECT
DISTRIBUTED OBJECT
REGULATORY AGENCY
ELECTRONIC FLIGHT STRIP
13
61
61
62
63
64
71
72
73
75
76
76
76
76
81
82
83
84
85

FIG. 7

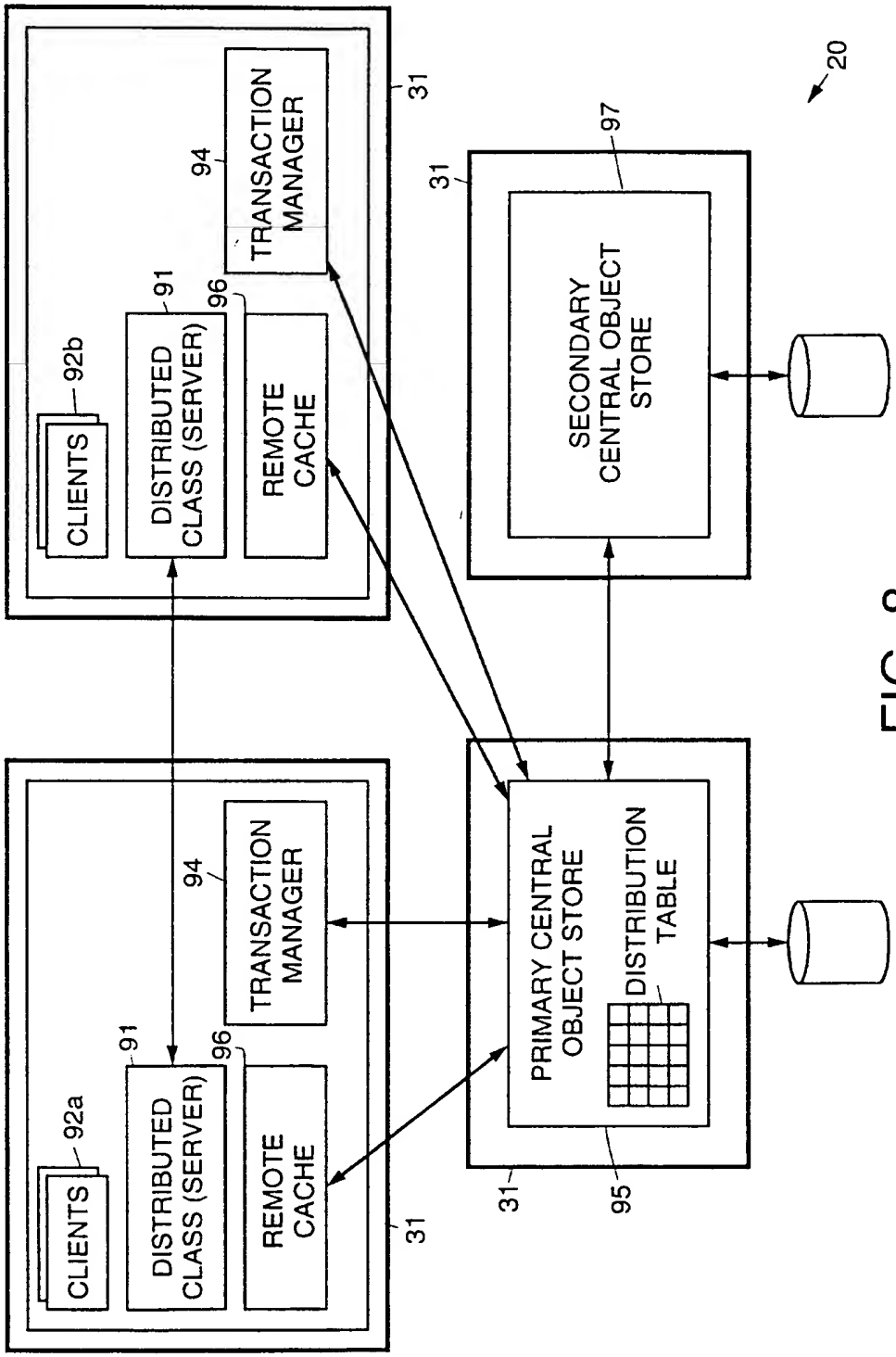

20
31
94
TRANSACTION MANAGER
91
96
DISTRIBUTED CLASS (SERVER)
REMOTE CACHE
92b
CLIENTS
97
SECONDARY CENTRAL OBJECT STORE
PRIMARY CENTRAL OBJECT STORE
DISTRIBUTION TABLE
95
92a

FIG. 8

INTERNATIONAL SEARCH REPORT

Inte. onal Application No
PCT/US 97/19304

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 G06F9/46

According to International Patent Classification(IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	LEA R ET AL: "COOL-2: an object oriented support platform built above the Chorus micro-kernel" PROCEEDINGS. 1991 INTERNATIONAL WORKSHOP ON OBJECT ORIENTATION IN OPERATING SYSTEMS (CAT. NO.91TH0392-1), PALO ALTO, CA, USA, 17-18 OCT. 1991, ISBN 0-8186-2265-2, 1991, LOS ALAMITOS, CA, USA, IEEE COMPUT. SOC. PRESS, USA, pages 68-72, XP002054687 see page 69, left-hand column, line 1 - page 70, left-hand column, line 26 ---	1,5-9
X	EP 0 524 077 A (ALCATEL NV ;CIT ALCATEL (FR)) 20 January 1993 see page 2, line 1 - page 3, line 25 --- -/--	1

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publicationdate of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

1

Date of the actual completion of theinternational search	Date of mailing of the international search report
5 February 1998	20/02/1998

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl. Fax: (+31-70) 340-3016	Brandt, J

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Inte. onal Application No
PCT/US 97/19304

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category '	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
A	"MODEL AND ARCHITECTURE FOR DIAGNOSTIC REQUESTS IN A HETEROGENEOUS DISTRIBUTED ENVIRONMENT" IBM TECHNICAL DISCLOSURE BULLETIN, vol. 34, no. 5, 1 October 1991, pages 451-455, XP000189819 see the whole document	1-9

1

INTERNATIONAL SEARCH REPORT

Information on patent family members

Inte onal Application No
PCT/US 97/19304

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
EP 0524077 A	20-01-93	FR 2679348 A	22-01-93
		CA 2073914 A	17-01-93
		JP 5204854 A	13-08-93